Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Master in Computing

Master Thesis

Personalizing web search and crawling from clickstream data

Student: Pau Vallès Fradera
Director: Ricard Gavaldà Mestre

Date: 19-01-2009

Contents

Chapter 1

Introduction

1.1 Web Search: origins, today's usage, problems

In the beginning, there was a complete directory of the whole World Web. These were the times when one could know all the existing servers in the web. Later, other web directories appeared. Some of them are Yahoo, Altavista, Lycos and Ask. These newer web directories kept a hierarchy of the web pages based on their topics. Web directories are human-edited, thus making them very hard to maintain when the web is growing up so fast. As a result, information retrieval techniques that had been developed for physical sets of documents, such as libraries, were put into practice in the web.

The first web search engines appeared on 1993. Those web search engines did not keep information about the content of the web pages, instead, they only indexed information about the title of the pages. It was in 1994, when web search engines started to index the whole web content, so that the user could search into the content of the web pages, not only in the title.

On 1998, Google appeared and this changed everything. The searches done by this search engine got better results than the previous search engines would get. This new search engine considered the links structure of the web, not only its contents. The algorithm used to analyze the links structure of the web was called PageRank. This algorithm introduced the concept of "citation" into the web: the more citations a web page has, the more important it is; furthermore, the more important is the one who cites, the more important the cited is. The information about the citations was taken from links in the web pages.

Nowadays, web search engines are widely used, and their usage is still growing. As of November 2008, Google performed 7.23 billion searches (according to [12]).

Web search engines are today used by everyone with access to computers, and those people have very different interests. But search engines always return the same result, regardless of who did the search. Search results could be im-

proved if more information about the user was considered.

1.2 Aim of the thesis

Our aim is to improve web search engines, approaching the searching problem considering the user, his/her topics of interest and the navigation context. Furthermore, the clickstream also contains patterns inside. Our system will also try to predict the next pages that are going to be visited according to the clickstream.

In a personalized search engine, two different users get different results for the same query, because the system considers the interests of each user separately. To personalize search, many sources of information can be used: the bookmarks of the user, his/her geographical location, his navigation history, etc.

Web search engines have, broadly speaking, three basic phases. They are crawling, indexing and searching. The information available about the users interest can be considered in some of those three phases, depending on its nature. Work on search personalization already exists. We will see them in Chapter 3.

In order to solve the problems of ignorance in relation to the user and his interests, we have developed a system that keeps track of the web pages that the user visits (his clickstream).

Our system will analyze the clickstream, and will focus the crawling to pages related to the topics of interest of the user. Furthermore, each time the user executes a query, the system will consider his/her navigation context, and pages related to the navigation context will get better scores.

Furthermore, our system also analyzes the clickstream of the user, and retrieves some navigation patterns from it. Those patterns will be used to give some navigation tips to the user based on his navigation context.

1.3 Thesis structure

In Chapter 2, we introduce the basis and the techniques related to search engines, emphasizing the most closely related to our thesis. In general terms, these are: some techniques to build search queries, some Page Ranking algorithms and some data mining topics, such as clustering and classification.

Chapter 3 presents the state of the art of the topics related to the thesis. In this chapter we analyze the existing research on the personalization of search engines, furthermore we also take a look at the existing research on navigational user patterns and “next page” prediction.

In Chapter 4 we present our work on the Personalized Search (PS) system. In this chapter we analyze how our system works. This is, we explain what algorithms we use to achieve our aim, their cost, and the improvements we have made to the original algorithms to fit our needs better.

In Chapter 5 we present the Navigation Tips program. We explain our purpose on making the Navigation Tips, how it works, and the algorithms we have used to achieve our interests.

In Chapter 6 we show the internal architecture of the PS system and the Navigation Tips. Furthermore, we explain how the different parts are connected.

In Chapter 7 we make an evaluation of our PS system and our Navigation Tips.

In Chapter 8 we present our conclusions and the future work.

The appendix contains a manual for the software we have developed.

Chapter 2

Background

2.1 Information Retrieval on the Web

Definition. Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text, but also images, videos, music ...) that satisfies an information need from within large collections (usually stored on computers).

For decades information retrieval was used by professional searchers, but nowadays hundreds of millions of people use information retrieval daily.

The field of IR also covers document clustering and document classification. Given a set of documents, clustering is the task of coming up with a good grouping of the documents based on their contents. Given a set of topics, and a set of documents, classification is the task of assigning each document to its most suitable topics, if any. IR systems can also be classified by the scale on which they operate. Three main scales exist:

- IR on the web.
- IR on the documents of an enterprise.
- IR on a personal computer.

When doing IR on the web, the IR system will have to retrieve information from billions of documents. Furthermore, the IR system will have to be aware of some webs, where its owners will manipulate it, so that their web can appear on the top results for some specific searches. Moreover, the indexing will have to filter, and index only the most important information, as it is impossible to store everything.

Also say that during the latest years, the Web 2.0 has emerged. Now web users do not only retrieve information from the web, but they also add value to the web. If the search engines are capable to retrieve implicit information (such as the number of visits), or explicit information (such as rankings) given by the users, they will get more accurate results.

When the IR is on a personal computer, IR will be based on the documents on this computer, and on its e-mails. In this case, the IR is a lot more personalized than the one done on the web. This allows the user to classify the documents according to his topics of interest. This classification will fit better his/her interests than the one done by a web search engine.

IR on an enterprise documents is in the middle of the other two cases. In this case, the IR system is placed on a separate computer. And the IR system will cover the internal documents of the enterprise.

Information retrieval is now an extremely wide field and we cannot cover more than the basics in this thesis. We describe in the next sections the most important concepts for our purposes. For more extensive coverage of IR, see for example the textbooks [19, 22, 2, 25, 21]

2.2 Searches by content

To do a search by content, the user will enter the query. The search engine will return the documents that satisfy the query. This feature can be achieved by reading the content on all of the documents, and search for the words specified by the user. The problem on this technique is that it requires a lot of computational effort at query time (linear time in relation with the length of the documents stored), which makes it infeasible.

The solution to this problem is to index the content of the files. If we build an inverted index prior to the searches, we will speed-up the searches. An inverted index is composed by a dictionary of terms, which will contain all the terms in the lexicon. Then, for each entry in the dictionary, we have a list of documents; this list contains references to the documents where the term appears. Each item in this list is called a posting, and so the list is called postings list. This index will make it very easy and fast to know whether a word appears or not into a document. Moreover, it will also be easy to retrieve the document frequency of a term from the index (the document frequency of a term is equivalent to counting the number of documents where this term appears in). The document frequency of the terms is very useful to retrieve information from a collection of documents.

Some additional information can be stored in the inverted index, for example, we can also store the term frequency for each posting (the term frequency of a term T in a document D is defined as the frequency of the term T in the document D).

These are the steps to follow to build the inverted index:

- Collect the documents to be indexed.
- Tokenize each text, turning each document into a list of tokens.
- Do a linguistic preprocessing: normalize each token to produce the list of indexing terms and drop unuseful tokens.

- Create the dictionary containing the indexing terms.
- Read each document to add its postings into the postings lists.

Once we have the inverted index, we will store the dictionary in main memory, meanwhile posting lists will be stored in the hard disk, as they are too large to be on main memory.

In the next example, we show how we can build an inverted index from a collection of documents. For our example we will consider the next collection of documents:

1. I have a black dog.
2. I have a white cat.
3. I like cats, so I have a cat.

Once we have the collection of documents, we have to tokenize each document, turning each document into a list of tokens:

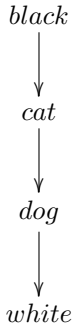
1. $I \longrightarrow have \longrightarrow a \longrightarrow black \longrightarrow dog$
2. $I \longrightarrow have \longrightarrow a \longrightarrow white \longrightarrow cat$
3. $I \longrightarrow like \longrightarrow cats \longrightarrow so \longrightarrow I \longrightarrow have \longrightarrow a \longrightarrow cat$

Now we have to do a preprocessing of the text, so that each token is normalized and unuseful tokens are suppressed. After the preprocessing, we get the following lists of tokens:

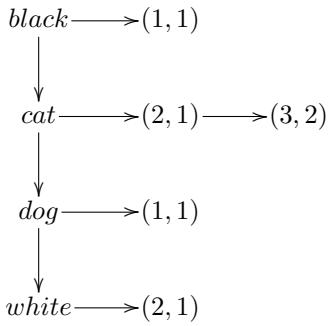
1. $black \longrightarrow dog$
2. $white \longrightarrow cat$
3. $cat \longrightarrow cat$

Note that useless terms like “I”, “have”, “a”, etc. have been removed from the tokens lists, as they give no hints about the content of the document. Furtermore, the term “cats” in document 3 has been replaced by “cat” during the normalization step, so that it is possible to find a correlation among documents that contain the term “cat” and documents that contain the term “cats”.

From the previous lists of tokens, we build the dictionary of terms:



Finally, we add the postings to their respective term (in this case, we have added information about the term frequency, so each posting has the form $(document, termfrequency)$):



2.2.1 Boolean Retrieval Model

Boolean queries only can express the appearance or not appearance of some terms in a document. This model of queries is very limited, and cannot rank the results: a document satisfies the query or it does not, but there is no middle course. Extended Boolean Model is similar to the Boolean Retrieval Model, but with some additional operators as term proximity operators. The Extended Boolean Model was the most used during the early 90's.

2.2.2 Ranked Retrieval Model

Ranked Retrieval Model is more complex than the Boolean Retrieval Model, and allows the user to execute queries in free text (without boolean or proximity operators). This feature makes Ranked Retrieval Model more user-friendly than Boolean Retrieval Model and Extended Boolean Model. Furthermore, the results of the search are ranked by score, so that the most representative documents of the search will appear on the top of the results. Furthermore,

structure of a Ranked Retrieval Model query is the same as the structure of a document, this makes it possible to have a similarity function that works for both: document-query comparisons and document-document comparisons. This function cannot exist for the Boolean Retrieval Model, as a boolean query structure and a document structure are different.

Nowadays, Ranked Retrieval Model is the most used. Ranked Retrieval Model queries do not need to have any boolean operators, which makes them more user-friendly than boolean queries. Furthermore, a plain-text Ranked Retrieval Model query will usually return a good result. However, Ranked Retrieval Model queries may not be enough in some cases. Therefore search engines also allow the execution of boolean queries when using the "Advanced Search" option, as using boolean operators in the queries can help to get a more selective result. This makes boolean queries specially useful when the user knows what he/she is looking for.

2.3 Crawling

A search engine needs to have an index containing information about a set of web pages. Before indexing the documents, we need to have the documents. The component that will provide the documents and their content is the crawler.

The crawler will surf the Internet, or a part of it, searching for the most interesting web pages. The interesting pages will be stored locally, so that they can be indexed later. The crawler is also known as bot or spider.

Crawlers can be classified as focused or unfocused. Unfocused crawlers store information about a page, regardless of its topic and its site; unfocused crawling is used by large-scale web search engines. By contrast, focused crawlers store information only of some of the web pages, thus focusing their crawling on some topics, or some sites, or some type of documents, or to interesting pages according to information retrieved from the user.

In our case, the we have a focused crawler that uses information from the user to focus the crawling.

2.4 Page Ranking

The purpose of Page Ranking is to measure the relative importance of the pages in the web. There are many algorithms for this purpose. The most important ones are: Hyper Search, Hyperlink-Induced Topic Search (HITS), PageRank, TrustRank, and OPIC.

2.4.1 Hyper Search

Hyper Search has been the first published technique to measure the importance of the pages in the web. This algorithm served as a base for the next ones. For more information about Hyper Search refer to [20].

2.4.2 Hyperlink-Induced Topic Search

HITS algorithm, also known as Hubs and Authorities, is a link analysis algorithm for the web. It is executed at query time and is used to modify the ranking of the results of a search by analyzing the link structure of the pages that will appear in the result of the search.

HITS algorithm assigns two different values to each web page: its authority value, and its hub value.

The authority value of a page represents the value of the content in the page, meanwhile the hub value estimates the value of its links to other pages.

The first step in the HITS algorithm, is to retrieve the set of pages in the result of the search, as the HITS algorithm only analyzes the structure of the pages in the output of the search, instead of all the web pages.

Then the authority and hub values for each page are set to 1. Next, the algorithm performs the following iteration as many times as necessary:

- Update the authority value for each page. The authority value of a page will be computed as the sum of the hub value of all the pages that point to it. This will give high authority value to pages that are pointed by pages that are recognized as hubs.
- Update the hub value for each page. The hub value of a page will be computed as the sum of the authority value of all the pages it points to. This will recognize as hub pages the ones that point to authority pages, that is, the pages that point to pages with relevant content, will be recognized as hubs.
- Finally all authority and hubs values will be normalized. This is achieved by dividing each authority score by the total of the authorities score, and diving each hub score by the total of the hubs score.

You can find more information about the HITS algorithm in [15].

2.4.3 PageRank

PageRank is a link analysis algorithm to measure the page relevance in a hyperlinked set of documents, such as the World Wide Web. This algorithm assigns a numerical weight to each document. This numerical weight is also called PageRank of the document.

The PageRank of a web page represents the likelihood that a person randomly clicking will arrive at this page.

The PageRank algorithms requires several iterations to be executed. At each iteration, the values will be better approximated to the real value.

In its simplest form, PageRank uses the next formula for each web page at each iteration:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

Where u is a web page, B_u is the set of pages that link to u, $PR(v)$ is the PageRank of v, and $L(v)$ is the number of out-links in page v.

At each iteration, the $PR(u)$ of each page u will be updated according to the values of $PR(v)$ in the latest iteration. After several iterations, the value contained in $PR(u)$ will be a good approximation to its real value.

For some weird structures of the links, the PageRank algorithm explained above may not converge or may have several different solutions. Solutions to this problem exist, but they are not explained in this thesis, as they are out of the scope of the thesis. For more information about the PageRank algorithm, please refer to [4, 22].

2.4.4 TrustRank

TrustRank is a semi-automatic link analysis technique for separating useful webpages from spam.

The HITS and the PageRank algorithms can also be used for this purpose, but they have been subject to manipulation.

To avoid this manipulation, the TrustRank algorithm selects a small set of documents. This set of documents will be evaluated by an expert. Once the reputable seed pages are manually identified, a crawl extending outward from the seed set seeks out similarly reliable and trustworthy pages.

For more information about TrustRank, please refer to [7].

2.4.5 On-line Page Importance Computation

The On-line Page Importance Computation algorithm, also known as OPIC, is a link analysis algorithm. Its capability to compute the page importance without storing the whole links graph is what makes it different from other algorithms.

We next explain how the OPIC algorithm works:
The OPIC algorithm keeps 2 values for each page: its *cash*, and its *history*. Initially, some *cash* is distributed uniformly among all the nodes, for the case of N nodes, $\frac{1}{N}$ *cash* will be assigned to each node. The *cash* of a web page contains the sum of the *cash* obtained by the page since the last time it was crawled. The *history* of the page contains the sum of the *cash* that the page has obtained since the algorithm started until the latest time it was crawled. When a page p is retrieved, its *cash* is added to its *history*. Moreover, its cash is distributed uniformly among all of its children, in other words:

$$p.history = history(p) + cash(p)$$

$$\forall p', p' \in children(p) \Rightarrow p'.cash = cash(p') + \frac{cash(p)}{size(children(p))}$$

$$p.cash = 0$$

For more information about the OPIC algorithm, please refer to [1].

2.5 Data Mining

Data mining is the process of retrieving patterns and useful information from data. For information about data mining, please see [23, 14, 13].

In the context of web search, some data mining techniques are more applicable than others. The most used data mining techniques in the field of web search are: document clustering and document classification techniques. Many of these applications are based in the Vector Space Model. In the next sections we explain the Vector Space Model and the main machine learning and data mining techniques used in the field of information retrieval. For more information about these techniques, please refer to [19, 22].

2.5.1 Vector Space Model

Search engines return the result in a ranking, that is, the documents that best fit the query appear in the top positions. To perform such a ranking, we need a function to measure the query-document similarity. For this purpose, usually each document is transformed to a vector, then a vector distance function will be used.

A common technique to transform a document into a vector is tf-idf, but many more techniques for this purpose exist. These techniques are based in two assumptions:

- The more frequent a term t is in a document d, the more relevant t is in d.
- The more frequent a term t is among a set of document D, the less relevant t is in d.

Given a set of documents D, the tf-idf technique transforms a document d into a vector v, where v has one component for each term that appears in d.

$$v_t = weight_{d,t} = tf_{d,t} * idf_t$$

Where $tf_{d,t}$ is the term frequency of the term t in the document d, that is, how many times t appears in d; and idf_t is the inverse document frequency of t in D. idf_t is computed as:

$$idf_t = log\left(\frac{\mid d' \in D : t' \in d \mid}{\mid D \mid}\right)$$

Then, the cosine distance C, or the Euclidean distance E can used to compute the similarity between two vectors. The following formules are used to compute the cosine distance and the Euclidean distance.

$$\begin{aligned} C &= cosineDistance(\vec{c},\vec{d}) = arccos\left(\frac{\vec{c}\times\vec{d}}{\mid\vec{c}\mid\mid\vec{d}\mid}\right) \\ &= arccos\left(\frac{\sum_{t\in(terms(c)\bigcap terms(d))} c_t \times d_t}{\mid\vec{c}\mid\mid\vec{d}\mid}\right) \end{aligned}$$

$$E = euclideanDistance(\vec{c},\vec{d}) = \sqrt{\sum_{t\in(terms(c)\bigcup terms(d))} (c_t - d_t)^2}$$

This is, after all, a heuristic, but has been intensely studied from theoretical and experimental viewpoints and has become a de facto standard.

When using the Vector Space Model, we will represent each document as a vector in the vector space model. In Vector Space Model each document is represented as a vector with one real-valued component, usually a tf-idf weight, for each term. Thus, the document space $\mathbb{X}$, the domain of the classification function, is $\mathbb{R}^{|V|}$.

Contiguity hypothesis. Documents in the same class form a contiguous region and regions of different classes do not overlap.

Many clustering and classification algorithms use the Vector Space Model to represent documents, therefore, they lie on the contiguity hypothesis. Some of those algorithms are: K-Means, K Nearest Neighbours and Support Vector Machines.

2.5.2 Clustering

Clustering is the partitioning of datasets into clusters, so that clusters are coherent internally, but clearly different from each other.

There are many kinds of clustering algorithms. The most important ones are: hierarchical clustering and flat clustering.

2.5.2.1 Hierarchical Clustering

Hierarchical clustering builds up a hierarchy of groups by continuously merging the two most similar groups. Each of these groups starts as a single item. In each iteration, the similarity between each pair of groups is computed, so that the most similar pair of groups is merged together as a new group.

After several iterations (as many as initial number of items 1), all the items belong to the same group.

Finally, after hierarchical clustering is completed, the result can be viewed as a binary tree.

2.5.2.2 Flat Clustering

Flat clustering algorithms determine all the clusters at once.

K-Means is the most important flat clustering algorithm. This algorithms uses the Vector Space Model to represent the documents. Its objective is to minimize the average squared Euclidean distance of documents from their cluster centers where a cluster center is defined as the mean or *centroid* $\vec{\mu}$ of the documents in a cluster w:

$$\vec{\mu}\ (w) = \frac{1}{\mid w \mid} \sum_{\vec{x} \in w} \vec{x}$$

The definition assumes that the documents are represented as length-normalized vectors.

A measure of how well the centroids represent the members of their clusters is the Residual Sum of Squares (RSS), the squared distance of each vector from its centroid summed over all vectors:

$$RSS_k = \sum_{\vec{x} \in w_k} \mid \vec{x} - \vec{\mu}\ (w_k) \mid^2$$

$$RSS = \sum_{k=1}^{K} RSS_k$$

K-Means algorithm:

The first step of K-means is to select as initial cluster centers K randomly selected documents (where K has been provided by the user), the seeds. The algorithm then moves the cluster centers around in space in order to minimize RSS. This is done by repeating iteratively two steps:

1. Reassign each document to the cluster with a nearest centroid.
2. Re-compute the centroid for each cluster (based on the documents belonging to the cluster).

This iteration is repeated until a stopping criterion is met. The most usual stopping conditions are:

- A fixed number of iterations has been completed.
- Assignment of clusters does not change between iterations.
- Terminate when RSS falls below a threshold. In practice, we need to combine it with a bound on the number of iterations to guarantee termination.
- Terminate when the decrease in RSS falls below a threshold, which indicates we are close to convergence. Again, we need to combine it with a bound on the number of iterations to prevent very long runtimes.

In our application, we have used a K-Means variation as text clustering algorithm.

2.5.3 Classification

In text classification, we are given a description $d \in \mathbb{X}$ of a document, where $\mathbb{X}$ is the *documentspace*; and a fixed set of *classes* $\mathbb{C} = \{c_1, c_2, ..., c_J\}$. We are given a training set $\mathbb{D}$ of labeled documents $\langle d, c\rangle$, where $\langle d, c\rangle \in \mathbb{X} \times \mathbb{C}$

Using a *learning method* or a *learning algorithm*, we then wish to learn a *classifier* or a *classification function* that maps documents to classes.

This type of learning is called supervised learning because the supervisor (the human who defines the classes and labels training documents) serves as a teacher directing the learning process.

2.5.3.1 Naïve Bayes

Naïve Bayes is a probability-based classification algorithm. The probability of a document d being in class c is computed as:

$$P\left(c \mid d\right) \propto P\left(c\right) \prod_{1 \leq k \leq n_d} P(t_k \mid c)$$

where $P\left(t_k \mid c\right)$ is the conditional probability of term t_k occurring in a document of class c. We interpret $P\left(t_k \mid c\right)$ as a measure of how much evidence t_k contributes that c is the correct class. $P\left(c\right)$ is the prior probability of a document occurring in class c. If a document's terms do not provide clear evidence for one class versus another, we choose the one that has a higher prior probability.
Our goal is to find the best class for the document, so we will choose the class for which the highest probability is estimated.

2.5.3.2 Support Vector Machines

Support Vector Machines, also known as SVMs, are a set of supervised learning methods used for classification. SVMs are not necessarily better than other machine-learning methods (except perhaps in situations with few training data), but they perform at the state-of-the-art level and have much current theoretical and empirical appeal.

SVMs use the Vector Space Classification to represent documents. Once a set of training documents is provided by the user, the algorithm will try to find hyperplanes in an extended feature space that separate the different classes.

SVMs are very complex, but we just explain its basis, as deeper details are out of the scope of this thesis. For further details on SVMs see [19].

2.5.3.3 K-Nearest Neighbours

k-Nearest Neighbours is an algorithm that classifies objects based on their closest training samples in the feature space.

Given a collection of items C, an item I, and a natural number K, the k-Nearest Neighbours algorithm will return a collection of items S containing the K items in C that are closest to I. Thus, it is necessary to have a function $dist(a, b)$ which computes the distance between the items a and b based on their similarity (the more similar, the less distance).

If the selected K value is too large, it will return items that are too far from I, thus adding noise. On the other hand, if K is too low, the set of items returned may not be representative enough, and may introduce random variations.

The itemset returned by K Nearest Neighbours S can be used to classify I, or to do regression on I according to the items in S.

For more information about the k-Nearest Neighbours algorithm see [22].

Chapter 3

State of the art

3.1 User Navigation Patterns

There are many approaches for discovering patterns in the users navigation. For example, [6] introduces new algorithms to retrieve a taxonomy of a single web site from the click-streams of its users.

In [8] they have developed a system to find out how the time affects the user behavior while surfing a web page. That is, they segment the logs of navigation of the users in different time intervals; and then they find what time intervals really interfere with the users behavior.

The two papers presented previously try to find navigation patterns among long groups of users for a single site. In this thesis, we consider instead finding the navigation patterns from a single-user approach; moreover, we consider all the pages that the user visits, not a single site. Also note that the previously presented works have been designed to work on the server side of the Web; by contrast, our system has been designed to work on the client side.

3.2 Personalization on Web Search Results

Many projects exist whose aim is to improve web searches by taking into account the needs of each user separately, that is, personalizing the searches to the needs of the user.

Some of them are [18, 17]. In these projects, they use the search history of each user to assign each user to a set of categories. When this user submits a query, the search engine will add his categories to the query, so that his query is conducted to his topics of interest. The main differences between our project and these is that we do not consider the *search* history of the user, but his *navigation* history. Furthermore, our system will also consider the navigation context of the user, so that the result of the search is focused to the topics that the user is currently reading.

In [10] they do not use a single PageRank vector, but one PageRank vector for each topic (one page may be representative for a topic, but not for another). In our system, we do not use PageRank vectors to focus the search, but we modify the query. Unlike our system, [Topic-Sensitive PageRank] has no consideration for the navigation context of the user.

In [9] they consider the search from a mobile device. In this case, the fact that mobile devices do not have as good input/output system as normal desktop computers, makes it very important to get the desired search results in the top positions, what will avoid a waste of time and effort for the user. To accomplish this, the queries submitted by the user will be refined by a system. This refinement is based on the user's current geographical location and real-world context, which are inferred from its place-names and the Web. In our system, we also do a query refinement based on the context. But the context considered in our system is not based on the geographical location, it is based in the navigation context of the user instead. Furthermore, our system also takes into account the organization that the user has done previously.

The approach to context-awareness in web search presented in [16] is closer to our approach than the previously cited articles. In this article they consider the content of the web pages that the user is visiting and the content of the opened documents in the word processor. In our system, we also consider the content of the documents browsed by the user; but we do not analyze the content in the same manner. In [16], no classification about the interests of the user has been done before, so the content of the documents is the only information available; by contrast, our system already has information about the topics of interest of the user, and hence more information about each topic is available, not only the content of the current files. The information about the classification done by the user allows our system to classify the current documents to their correspondent topic, and hence retrieve the most important words of the topic itself, not only the words that appear in the current documents.

In [3] a web search engine is presented that works in a P2P environment. In this system, each user has his/her own crawler and searcher, but the indexes retrieved by the crawler are shared among all the peers. The crawler is a focused crawler, so that the crawling is based on the bookmarks of the user, thus focusing the crawling in his topics of interest. In our system, we also focus the crawling, but instead of using a fixed set of bookmarks, we use a dynamic set of web pages based on the navigation history of the user, thus changes in the interests of the user will be considered by the crawler automatically, without the need of a explicit mention (like changing the bookmarks) by the user. The crawler of this system is called Bingo!, and the searcher is called Minerva.

At the beginning of our work, we considered to expand Bingo! and Minerva to build the PS system. But we dismissed this option. The reasons for this decision are explained in Chapter 6.

Chapter 4

The PS system - Personalized Search

Most web search engines consider very little information about the user and his context, and so the result of a search is too generalized. In order to make more specific searches according to the user, his interests and his context, we have developed the PS system.

PS, short for Personalized Search, is our personalized search and crawling system. Some parts of PS work on Nutch, an existing open source search engine [11]. We chose Nutch because it has been designed to be extended, as it includes plugin system. PS is composed of three parts: Nutch Crawler, Organizer and Nutch Query.

The Organizer is a program that allows the user to build a hierarchy with the already crawled documents. This hierarchy can be built using a clustering algorithm or a classification algorithm. We offer two different algorithms to provide two different ways to organize the documents.

For the clustering algorithm, the user only has to provide information about the number of clusters, requiring little effort, and so time, for the user.

The classification algorithm requires more participation from the user, as he/she has to provide information about each class. The advantage of the classification algorithm versus the clustering algorithm, is that the resulting classification will be more personalized by the user, and so, it will fit his interests better than the clustering algorithm.

The document hierarchy obtained by one or the other algorithm will be used by Nutch Crawler and by Nutch Query.

The Nutch Crawler component performs the function of crawling the documents that can be of interest for the user. To achieve this, a hierarchy for the existing documents and the navigation history of the user are necessary.

The Nutch Query is a web application. This application is directly used to enter the search queries and get their results back.

4.1 Organizer

The task of the Organizer is to keep all the crawled documents organized in a tree hierarchy. The user has to choose one of the available organization methods. One of them is a clustering algorithm, and the other one is a classification algorithm.

At first all the documents belong the root cluster. Then the user can split any cluster into more clusters with the help of the organizer, thus building a tree hierarchy.

For the case of the clustering algorithm, the user has to provide the final number of clusters for the clustering before its execution. Then the clustering algorithm will try to find the clustering that best fits for the given set of documents and the number of clusters provided by the user.

For the case of the classification algorithm, when the user wants to split a cluster, he/she will have to provide a list of children clusters. Furthermore, for each child cluster, the user will have to specify the list of documents that best represents the cluster. These documents will be used as training data.

Also say that the Organizer also offers the possibility to blacklist the clusters that are not of his/her interest. This information will be used by the Nutch Crawler and by the Nutch Query.

4.1.1 Clustering

Text clustering requires a lot of computational effort. Sets of documents have, typically, hundreds of thousands of different terms, which makes distance computation very expensive.

This makes choosing an efficient algorithm very important. Hierarchical clustering algorithms are the ones that get the best clusterings, furthermore the result is shown as a tree (the hierarchy); but they are very expensive to run (at least n^2, where n is the number of documents). The execution cost is what makes them unfeasible.

Flat clustering algorithms are another option. In flat clustering algorithms, the user has to provide the number of clusters prior to the clustering, which makes them not as good as hierarchical clustering algorithms, but they have a cost proportional to n, where n is the number of documents. This made us to choose a flat clustering algorithm: we chose K-Means for ease of implementation (another option was k-medoids).

In K-Means the input is: a set of items, and the number of clusters; and the output is a clustering where each document has been assigned to one cluster.

In the next figure, we show the pseudo-code of the basic k-Means algorithm for a given x and K, where x is a set of items and K is the desired number of final clusters:

```
 1  var m = initialCentroids(x, K);
 2  var N = x.length;
 3  while (!stoppingCriteria) {
 4    var w = [][];
 5    // compute membership in clusters
 6    for (var n = 1; n <= N; n++) {
 7      var min_dist = infinity;
 8      for (var i = 1; i <= K, i++) {
 9         var dist = dist(m[i], x[n]);
10         if (dist < min_dist) {
11            min_dist = dist;
12            v = i;
13         }
14      }
15      w[v].add(n);
16    }
17    // recompute the centroids
18    for (var k = 1; k <= K; k++) {
19      m[k] = avg(x in w[k]);
20    }
21  }
22  return m;
```

This algorithm was explained in Chapter 2, but we show the pseudo-code again in order to discuss in detail its running time and the possible points where it can be optimized.

Lines 9 and 19 are the ones that consume most of the CPU execution time. On line 9, the distance between a document d and a cluster c is computed; this has to be done for every pair document-cluster. On line 19, the average of all the documents in a cluster is calculated, thus computing the position of the centroid. This part is also expensive in CPU time, so we will also try to improve it.

In K-Means, the Euclidean Distance is used to compute the distance between a document and a centroid. But in our implementation, we use the Cosine Distance instead. This is because the cosine distance computation is much faster. Furthermore, the resulting clustering will be the same, regardless the Euclidean Distance or the Cosine Distance is used (provided that the centroids and the document vectors are normalized).

To compute the Euclidean Distance we have to compute:

$$E = euclideanDistance(\vec{c}, \vec{d}) = \sqrt{\sum_{t \in (terms(c) \bigcup terms(d))} (c_t - d_t)^2}$$

But to compute the Cosine Distance we only have to compute:

$$\begin{aligned} C &= cosineDistance(\vec{c}, \vec{d}) = arccos\left(\frac{\vec{c} \times \vec{d}}{|\vec{c}||\vec{d}|}\right) \\ &= arccos\left(\frac{\sum_{t \in (terms(c) \bigcap terms(d))} c_t \times d_t}{|\vec{c}||\vec{d}|}\right) \end{aligned}$$

Where $\vec{c}$ is the vector that represents the position of the centroid c, and $\vec{d}$ is the vector that represents the position of the document d. $terms(c)$ represents the set of terms that appear in the centroid c, and $terms(d)$ represents the set of terms that appear in the document d. c_t and d_t represent the value of the t component in $\vec{c}$ and $\vec{d}$, respectively. If c_t or d_t is not defined for a given term t, then the value 0 will be used instead.

For the Euclidean Distance, we have to consider the union of terms between the centroid and the document, meanwhile for the Cosine Distance we only have to consider the intersection of terms between the centroid and the document (except for $|\vec{c}|$ and $|\vec{d}|$, but we will see this case later). In practice we will only iterate among the terms of the document (much less than the terms in the centroid).

For Cosine Distance calculus, we need to compute $|\vec{c}|$. To compute $|\vec{c}|$ we have to consider the whole $\vec{c}$ vector, not only the components that also appear in $\vec{d}$, but $|\vec{c}|$ will remain constant during each iteration, so it will only be computed at the beginning of each iteration. Similarly to $|\vec{c}|$, $|\vec{d}|$ will only be computed before the first iteration, as it will remain constant.

4.1.2 k-Means incremental clustering considerations

Our k-Means algorithm has to consider lots of documents for clustering, but not all documents are available from the start: new documents will appear each time a crawling is performed. If k-Means is executed after a crawling, we should use the existing clustering as a base for the new one. This incrementality can help to make clustering of large collections of documents cheaper.

Usually k-Means performs a random assignment of documents to clusters at the beginning. But if we have a previous clustering, we can start by assigning old documents to their previous clusters and assigning new documents to their nearest cluster.

This change in the k-Means algorithm will result in a great improvement in the execution time: the initial state of the clustering will be similar to the state of the original K-Means algorithm after several iterations, unless a large amount

of new documents with very different contents has been added. However, this is not probable for the nature of the source of documents.

The set of documents grows up at each crawling, so considering the whole set of documents for the clustering would make each clustering slower than the previous one. One way to avoid this increment in the cost of the clustering, is to consider only the new documents and a subset of fixed size of the old ones.

4.1.3 k-Means improvements

Since the application of k-Means algorithm is central to our project, and must work on large amounts of documents, we tried and tested several strategies for improving its efficiency. We report those that we found useful and we adopted, and one that, while promising in theory, we did not manage to apply successfully to our context.

4.1.3.1 Do stemming on English documents

The index that Nutch provides is not the index that the Organizer uses for crawling. The index used by the Organizer is the original index with some modifications.

One of the modifications that were used at the beginning was to do some stemming to English documents, as this provides better clustering results and a cheaper clustering (as the number of terms is reduced). But finally we have declined this option.

The reason is that not only documents in the modified index are going to be considered for clustering or classification: the documents that the user surfs have to be classified in real-time too, but we have no way to recognize the language in those documents, so no stemming algorithm can be applied to them. This would make stemmed documents very different from not stemmed documents, and so their comparisons would have no sense.

4.1.3.2 Building a kd-tree to speed-up document clustering

A kd-tree is a data structure used to store a finite set of points from a finite-dimensional space. In a kd-tree, each node is a point in a k-dimensional space. Every non-leaf node generates a hyperplane that divides the space into two sub-classes. Points to the left of the hyperplane represent the left sub-tree, and points to the right of the hyperplane represent the right sub-tree. Each non-leaf node is associated with a split dimension d (one of the k dimensions) and a split value v , so that the hyperplane is perpendicular to the dimension d vector and its d value is v.

According to [Accelerating Exact k-means Algorithms with Geometric Reasoning] building a kd-tree to store the items to be clustered can make clustering

faster in some cases. This happens because in a kd-tree, each leaf node n contains all the items in a hyper-rectangle h. If for the hyper-rectangle h, all the points in it have the same "nearest centroid" c, then all the items in h can be assigned to c, skipping many distance computations.

Given an hyper-rectangle h, a centroid c_1 and a centroid c_2. If for all the points in h their nearest centroid is c_1 then:

- Consider the point p_1, which is the point in h which is furthest from c_1.
- Consider the point p_2, which is the point in h which is closest to c_2.
- $dist(p_1, c_1) < dist(p_2, c_2)$. This condition is necessary to prove that all the points in h have c_1 as their nearest centroid.

When working with a very high number of dimensions (our case), provided that for most dimensions no range of values has been specified by the parent nodes, the range of distances from a centroid c to the points in h is very high, and so it is hardly never possible to prove that $dist(p_1, c_1) < dist(p_2, c_2)$, and there is no point in building a kd-tree.

Due to the troubles we had with the high dimensionality of our data, we finally dismissed the usage of a kd-tree to speed-up our clustering.

4.1.3.3 Keeping if a cluster has moved during the latest iteration

During the latest iterations, most clusters do not move. So if we know that a cluster has not moved during the latest iteration, we can avoid the computation of its mean and its module, as it remains untouched.

4.1.3.4 Keeping the sum of the positions of all the documents in a cluster

If we keep the sum of the positions of the documents in a cluster, we make addition and deletion of a document into a cluster a bit more expensive. But then the centroid computation for a cluster turns to be much faster: only one division per term in the cluster is needed.

Before this improvement, we needed one sum per term in every document and also all the divisions.

4.1.3.5 Keeping the distance that a cluster has moved during the latest iteration

If we keep the distance that the cluster has moved during the latest iteration, we can also keep a minimum known distance between a document and a cluster.

This minimum distance is given by the next formula:

If a minimum distance between a document and a cluster for the latest iteration is unknown, then the current minimum known distance has to be computed as the real distance, which requires a high CPU usage.

If we know a minimum distance for the latest iteration, then the current minimum distance between a cluster C and a document D is:

$$mkd(C, D, n) = mkd(C, D, n-1) - dm(C, n-1)$$

Where $mkd(C, D, i)$ is the minimum known distance between the centroid C and the document D in the i^{th} iteration, and $dm(C, i)$ is the distance moved by the centroid C during the i^{th} iteration.

One property of mkd is that $mkd(C, D, i) \leq distance(C, D, i)$ (where $distance(C, D, i)$ represents the real distance between a document D, a centroid C during the i^{th} iteration). Also note that computation of mkd is much cheaper than the computation of real distance.

If $mkd(C, D, n)$ gets lower than the distance between the document D and its nearest cluster $C2$, then the distance between D and C has to be recomputed as the real distance. Then if $distance(C, D) < distance(C2, D)$, document D will be reassigned to cluster C.

As a result of this improvement, the first iteration of the algorithm has a higher execution time, as the distance between each document and each cluster has to be saved in memory. But in the next iterations, the execution cost is much lower (in general, each iteration is faster than the previous one).

The number of computations of the real document-cluster distance *comp* , which has a great incidence on the cost of one iteration, gets lower iteration after iteration. The explanation to the drop of *comp*, is that as the distance that centroids move drops, the minimum known distance values will be kept closer to the real distance values.

Keeping the distance between each pair cluster-document can be expensive in terms of memory consumption, its cost is $\Theta(C \times D)$, where C represents the number of clusters and D represents the number of documents.

If memory consumption is a problem, a possible solution would be to consider only the distance to the M nearest clusters at a given iteration i (the value of i can vary during the execution of the algorithm, as at some iterations all the distances will be recomputed again, and a new matrix will be obtained). Then the cost in terms of memory consumption will be $\Theta(M \times D) = \Theta(D)$, as M is constant. Note that in this case, for a given document d and a cluster c, where $c \notin nearestClusters(d, M, i)$ (where $nearestClusters(d, M, i)$ returns the set of clusters with size M that are nearest to d at the i^{th} iteration), $dist(c, d)$ will not be computed, and hence d will not be assigned to c even if c is its nearest

cluster. This may change the resulting clustering, but it is not probable. Also consider that in the further iterations, a recomputation of the matrix may take place. In this case d could be assigned to c, making the clustering nearly as precise as the original k-Means algorithm.

4.1.3.6 Creating several threads to find the nearest cluster for the documents (and reassign if necessary)

Finding the nearest cluster for N documents, could be converted into N independent jobs. This makes it distributable. Making this job distributed will improve the efficiency of the algorithm provided that it is executed in a multi-core or a multi-processor environment.

In our algorithm, M independent threads will be created, and the documents will be assigned randomly to one of the M threads.

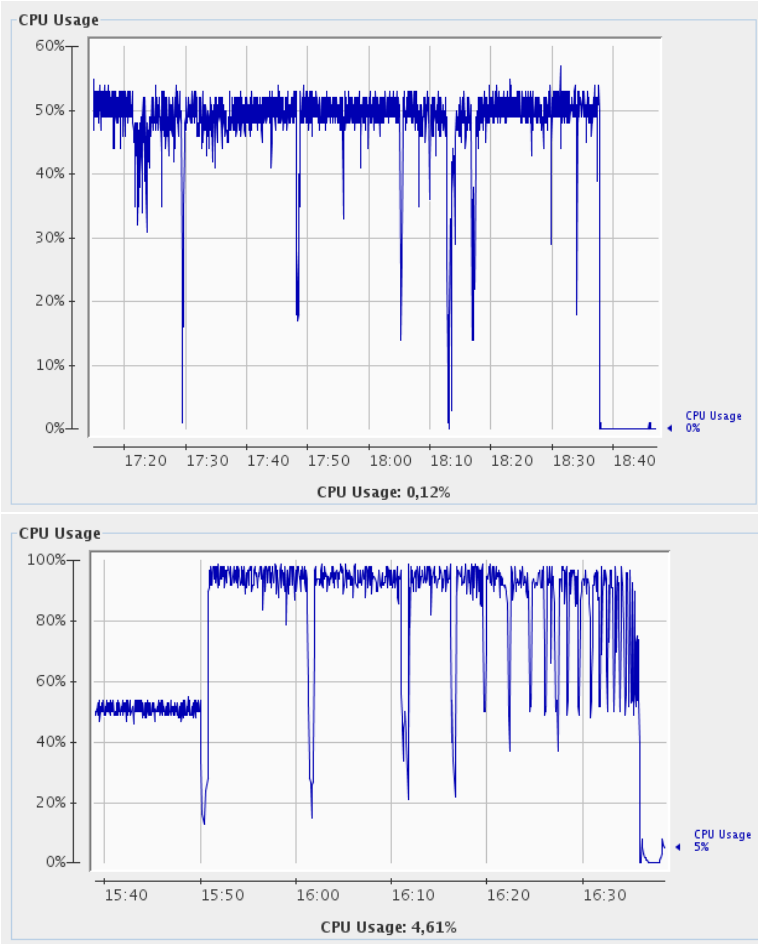
Similar documents take a similar time to compute their nearest cluster. Typically, similar documents are close each other in the index. If we assigned the consecutive documents to the same thread, we may create threads that would take very long to execute compared to other threads. To avoid this, our algorithm assigns each document to a random thread.

Finally, all threads will be executed concurrently.

The number of alive threads will always be kept below a constant P, so that CPU does not get saturated.

In our case, we have set $M = 20$ and $P = 5$. These values have reported to be good in Core2Duo processors, but their performance may vary depending on the hardware of the machine. Specially, the higher number of cores or processors, the higher M and P values should be used.

The next graphics, show the CPU usage for two executions of the clustering algorithm on the same data set:



The set of documents and the number of clusters was the same for both cases. The difference was on the P value used. The picture on the top shows the CPU usage when P is set to 1. The picture on the bottom shows the CPU usage when P is set to 5. As can be observed, the CPU usage grows up when P is set to 5, thus performing the whole work faster. Furthermore, for the case with $P = 1$, the first iteration lasted for 1125 seconds, meanwhile for the case with $P = 5$, the first iteration lasted for 680 seconds.

4.1.4 Classification

The Organizer also provides a classification system. The user can choose if the program will work using the results of the clustering algorithm or with the results of the classification algorithm.

Classification algorithms require more participation from the user than clustering algorithms, but they also provide better results.

Our implementation uses the Naïve Bayes algorithm. The user has to specify a set of classes for the classification, and set the most relevant document for

each class (those documents will be used as trademarks). Once each class has a few trademarks, the user will execute the reclassification. Then, each document will be reassigned to the class where it fits best.

The Naïve Bayes classification algorithm is faster than the K-Means clustering algorithm.

In Naïve Bayes, the probability of a document D to belong to a class C is computed as:

$$P(D \in C) = \frac{1}{Z} P(C) \prod_{t \in D} P(t|C)$$

Where $t \in D$ represents the set of terms that appear in D, and Z is a value that only depends on the terms that appear in D, thus making it constant, and hence ignorable for a probability comparison, when computing the probability of a single document to belong to different classes.

But in the implementation we have used:

$$P(D \in C) = log(P(C)) + \sum_{t \in D} log(P(t|C))$$

Because this function is strictly growing in respect to the original, but will not give underflow values.

Finally, a document d will be assigned to a class C, so that $P(D \in C)$ is maximum.

4.2 Nutch Focused Crawling

Our Crawling system uses two stand-alone programs and the Nutch Crawler with an extension. This makes the crawling focused on the preferred topics by the user, based on the pages that he/she has visited since the latest crawling and the already crawled pages.

To do a crawling, first a program called "CrawlUnknownHistoryURLs" has to be executed. This program will produce a list of URLs that appear in the user "latest navigation history", but have not been crawled yet. From this list of URLs, a crawling will be executed. This will make these URLs appear in the index.

The "latest navigation history" is the set of URLs that have been visited since the latest complete crawling cycle was performed.

Then the second stand-alone program, "ClusterHistoryURLs", will be executed. The function of this program is to do a clustering with the URLs in the "latest navigation history". Once this clustering has been performed, the

most significant documents will be retrieved and will be used as seeds for the definitive clustering.

"ClusterHistoryURLs" will perform a clustering with the "latest navigation history" URLs. This clustering will be a tree hierarchy clustering, where the tree has a depth of 2. First a clustering with all the document in the "latest navigation history" will be performed, the set of all the documents will be splitted into k clusters, where k is an internal parameter. By default $k = 10$. Then, each resulting cluster c that has a size greater than n will be splitted into m clusters. The n value is determined by the next formula:

$$n = \frac{docs}{3k}$$

Where *docs* is the total number of documents in the "latest navigation history".

For a cluster c, the m value is:

$$m = l * \frac{size(c) * k}{docs}$$

Where l is an internal parameter. A greater l value will result in more clusters in the tree, thus getting a higher number of URLs to be used as seeds. We have used $l = 4$ for our implementation.

Once the clustering has finished, the most representative documents in each leaf cluster c will be added to the final set of seed URLs S.

Finally, the definitive crawling will be executed with S as the set of seed URLs. Our system will focus the crawling to the documents that can be of interest for the user. To accomplish this, our system will consider the in-links of the candidate documents to be crawled: the more interesting are the inlinks, the more interesting the document is.

Computation of the interest of a document D to the user:

1. Given the nearest cluster C for the document D. If C is blacklisted, then its interest is -1.
2. If the cluster C is not blacklisted, then the interest for the document D is:
   (a) $1/(1 + distance(C, D))$ for the case of K-Means selected as the organizing algorithm.
   (b) 1 for the case of Naïve Bayes as selected as the organizing algorithm.

The score of a candidate document will be computed as the interest of the most interesting inlink. This score, will be added to the score provided by the original Nutch Scoring Filter, which uses the OPIC algorithm to compute the page importance.

After each crawling, an index is obtained. This index has to be modified in order to do some cluster with it. These are the steps we follow:

1. Keep only documents written in the languages preferred by the user.

2. Delete all words that appear in very few documents (less that 1/1000). We perform this step to reduce the size of the terms set size, as considering infrequent terms will not improve the clustering or classification, so they are a waste of memory and execution time in the further steps such as clustering and classification.

Remark: This functionality has been completely implemented inside Nutch, but, unfortunately, at the moment of closing this report, it is not working. In fact, Nutch skips the code corresponding to this path: we suspect it is due to an incorrect interfacing between the Nutch core and our plugin, that we have not been able to identify as of now. Let us remark, however, that the crawling performed by our system is still focused, to some extent, since the seeds are chosen from the user's history. We will make all efforts to solve this problem in the near future, to improve even further the system's focusing.

Once these steps have been performed, the new index is saved in another location, so that it is usable by the Organizer.

At first we considered the step of doing stemming to the documents in the index, as at this moment the language of each document is known, but finally we declined this option. Doing stemming would have resulted in a slight improvement on the clustering and classification results, furthermore the execution time would also have a slight drop, as the set of terms to consider would be shorter.

The reason why we declined the option of doing stemming is that we do not only classify documents in this index, but we also have to classify documents surfed by the user in real-time. The language of the documents that are being visited is unknown, so we do not know what stemming has to be applied to those documents. If those documents are not stemmed before being classified, they will be very different from the stemmed documents, and so the classification results will not be reliable.

4.3 Nutch Context-Aware Search

When a search is performed, the navigation context of the user is meaningful to the result of the search. When a user is visiting pages related to a topic T, probably the search is focused to that topic.

To take into account the navigation context, our system considers the latest pages that have been visited by the user. For each page visited by the user, the system will compute its nearest leaf cluster.

When the user submits a query, the system will modify it, so that it incorporates information about the navigation context. To achieve this, our system considers the clusters at which belong the latest pages (five by default) visited by the user.

Consider a hashtable *clusters* where each key is a cluster, and each value is a weight. The weight of a cluster represents the percentage of the latest pages visited that belong to this cluster.

Then a hashtable *terms* where each key is a term and each value is a weight will be created from the *clusters* hashtable. For each cluster that appears in the *clusters* hashtable, a set of the most meaningful terms will be retrieved. The weight of a term t will be computed as:

$$weight(t) = \sum_{c:t \in meaningfulTerms(c)} weight(c) * weight(t, c)$$

Where $weight(c)$ represents the percentage of documents in the "latest history" that have c as their nearest cluster, and $weight(t, c)$ represents the importance that the term t has in the cluster c. The computation of $weight(t, c)$ will vary depending on the organizing algorithm used:

- If k-Means is the organizing algorithm, then the importance of a term will be proportional to the Inverse Document Frequency of the given term.
- If Naïve Bayes is the organizing algorithm, then the importance of a term t in a cluster c over a set of clusters C will be computed as:

  $$weight(t, c) = \frac{fa(t, c)}{\sum_{c \in C} fa(t, c) + \alpha}$$

  Where $fa(t, c)$ is the proportion of documents in c that contain the term t. Finally, the importance of a term t will be computed as:

  $$weight(t) = \sum_{c \in C} weight(t, c)$$

  The $weight(t)$ is computed when the classification is performed, and thus it does not slow down the searches.

Chapter 5

Navigation Tips

World Wide Web users tend to describe some navigation patterns while surfing the web. The information of the user navigation history can be used to find out those patterns. Those patterns are helpful to predict the next page that the user is going to visit, and so give him/her some tips. Navigation Tips has been designed for this purpose.

Navigation Tips is composed by a Java program (HistoryAnalysis), and a Firefox extension (NTExt).

In Navigation Tips, there is a file called history.txt. This file contains information about the navigation history of the user. Furthermore, there is another file called rules.txt which contains the set of rules that have been retrieved from the history.txt file.

NTExt keeps a queue with the latest N URLs visited. From this queue and the rules contained in the rules.txt file, NTExt builds the navigation tips.

5.1 HistoryAnalysis

HistoryAnalysis function is to update the set of rules according to the navigation history of the user. That is, update the rules.txt file according to the history.txt file content.

5.1.1 History Analysis Algorithm

To achieve the purpose of HistoryAnalysis, we have implemented an algorithm to analyze the navigation history of the user, and retrieve the navigation rules from it. This algorithm works in the following way:

At first we have an empty set S of rule candidates, where a rule candidate R has the form $\langle from, to \rangle$, where $from$ is a URL, and to is a set of tuples of

the form $\langle url, weight \rangle$.

Then, many analysis of the history will be executed, and the rules extracted from them will be added to the set of rule candidates S. Each of the analysis is of the form: $analyze(history, weight, interval)$, where $history$ is a sequence of items of the form $\langle url, timestamp \rangle$ that represent a visit to the URL url at the instant $timestamp$, $weight$ is the weight that will be assigned to the rules derived from this analysis, and $interval$ is a relative time interval.

For a given $history$, our algorithm iterates over all the tuples contained in it. For each tuple $t = \langle url, date \rangle$ in $history$, consider the set of tuples T so that:

$$\forall t' \in history,\ t' \in T \quad \text{iff} \quad t'.date \in (t.date + interval)$$

Then, for each t' in T, a rule of the form $\langle from, \langle to, weight \rangle \rangle$ with the values $\langle t, \langle t', weight \rangle \rangle$ will be created. If the set S already contains a rule candidate $r = \langle from, to \rangle$ so that $r.from = t.url$ and $\exists to \in r.to, to.url = r.to$, then $to.weight = to.weight + weight$, else the rule r will be added to the set of rule candidates S.

Once all the analysis have been executed, the final set of rules $S2$ will be built from the rule candidates set S. For each rule candidate $R = \langle from, to \rangle$, $R \in S$,

where to is a set of tuples of the form $\langle url, weight \rangle$, a new rule $R2 = \langle from, to \rangle$ will be created, where $R2.from = R.from, R2.to = top5(R.to)$. $top5(R.to)$ is a function that returns the 5 tuples in $R.to$ that have the highest $weight$ and also have a minimum have a minimum proportion of $weight$ in relation to the total $weight$ of all the tuples in $R.to$.

Finally, the set of rules $R2$ is written to the file rules.txt. These rules will be used by NTExt.

The default analysis are: $analyze(longHistory, 1, [-200seconds, 600seconds])$ and $analyze(shortHistory, 2, [-200seconds, 600seconds])$. Where $longHistory$ is a sequence of visits that contains the latest 10,000 visits, and $shortHistory$ contains the latest 1,000 visits. As you can see, a higher weight will be assigned to the rules derived from the earliest visits, thus changes in the user navigation patterns will be detected soon. Furthermore, up to 10,000 visits are considered, so that rarely surfed pages, also appear in the rules. The time interval is $[-200seconds, 600seconds]$; this means that when a page P is visited at time t and a page P' is visited at time t', the rule $P \rightarrow P'$ will be considered iif $t' \geq t - 200$ and $t' \leq t + 600$, that is, page P' was visited in the interval $[-200seconds, 600seconds]$ relative to the time at which P was visited.

This algorithm is inspired (but a simple, specialized case) in algorithms for the so-called *frequent sequence mining problem*. In fact, we considered using the ISSA software for this purpose [5], that implements relatively algorithms for pattern mining. However, the generality of the capabilities of ISSA has a hight computational cost which is unnecessary for the problem that we want

to resolve. The input for the ISSA system is a set of sequences. From this set of sequences, ISSA will retrieve the sequences that have minimum support (given by the user). The first problem we found is that ISSA needs a set of sequences, but we only have one sequence: the *history*, thus we would need to split this sequence into more sequences to get the set of sequences. But how should we split the sequences? Moreover, ISSA only considers sequences, but gives no way to consider the time between different clicks. Finally, the execution of ISSA was too expensive, as we could only analyze sequences of up to 500 elements in a feasible time. For this reasons we finally dismissed using ISSA.

5.2 NTExt

NTExt, short for Navigation Tips Extension, is a Firefox Extension. Its purpose is to log all the pages visited by the user, that is, writing the navigation history into the history.txt file. Furthermore, NTExt will give navigation tips to the user according to his short term navigation history and to the rules contained in the rules.txt file.

Each time the user visits a page P at a time t, if the user has checked the option to "Log History", NTExt will write at the end of the file history.txt a new line containing the timestamp t and the URL of the page P.

Chapter 6

PS software architecture

6.1 PS System

6.1.1 Nutch Crawling and Nutch Search

Our aim was to build a search engine that considered more personalizations than current search engines. Thus, we needed an existing search engine that would be used as a base. Then, we would implement the additional features to make it more personalizable.

At first, we thought of extending the Bingo! and Minerva system [3], already mentioned in Chapter 3. Where Bingo! is a focused crawler, and Minerva is a P2P search engine. But we declined this option. We took this decision because we had spent several weeks trying to understand the existing code of the programs but it was not possible. Furthermore, there was no documentation about its architecture, what made it very hard to extend. We also were in contact with some of the developers of the project who eventually offered some help, but we were unable to advance.

As we could not advance our project, we decided to search for another solution different than Bingo! and Minerva. Finally, we found Nutch, an open source search engine designed to be easy to extend.

The Nutch Crawling and the Nutch Search systems have been mounted on the original Nutch program. Nutch is an open source search engine based on Lucene (an open source information retrieval library), which Nutch uses to index the crawled documents. An important feature of Nutch is that it allows the addition of features through plugins. The plugins system fits our needs, as we need an extendable search engine.

The plugins system has made it easier to add the additional features we needed. We have implemented a new "Scoring Filter" plugin for the Nutch Crawling, and a new "Context Query Filter" plugin for the Nutch Search.

6.1.1.1 Nutch Crawling

The crawling system has two standalone executable programs. Their function is to analyze the latest click-stream of the user, and build the set of URLs that will be used as seeds by the Nutch Crawler. They are CrawlUnknownHistoryURLs and ClusterHistoryURLs. The Nutch Crawling also uses the Crawling implement in Nutch in addition to our Nutch Extension.

When doing a new crawling these are the steps to follow:

1. Execute CrawlUnknownHistoryURLs. This program will build a list of all the URLs in the latest history (history since the last time that this program was executed) that have never been crawled.
2. Using the list of URLs provided by CrawlUnknownHistoryURLs, a crawling will be executed. This crawling will do these URLs appear in the index.
3. Execute ClusterHistoryURLs. This program will do a clustering of the latest history URLs (history since the last time that this program was executed). This clustering is possible because now the latest history URLs already appear in the index. Once the clustering has been done, this program will provide a list of the URLs that are close to their respective centroids.
4. A new crawling will be executed. In this crawling the list of URLs provided by ClusterHistoryURLs will be used as seeds.

6.1.1.2 Nutch Search

The Nutch Search system also has an external executable program. This program is called UpdateContext. Its function is to analyze the documents that are being visited by the user, and update the "navigation context" according to the documents visited by the user. That is, finding the nearest leaf cluster for each page that the user visits, and inserting this information into the database.

The information about the "navigation context" will be used by the "Context Query Filter" plugin. This plugin will modify the queries submitted by the user according to the "navigation context", thus adding information about the navigation context of the user.

Note that the computation of the nearest cluster for a web page is done while the user is surfing that page, not while the user submits the query. Therefore, computing the nearest cluster for a page does not add time to the search execution. Thus, only the computation of the weight of context-related terms adds extra time to the query execution.

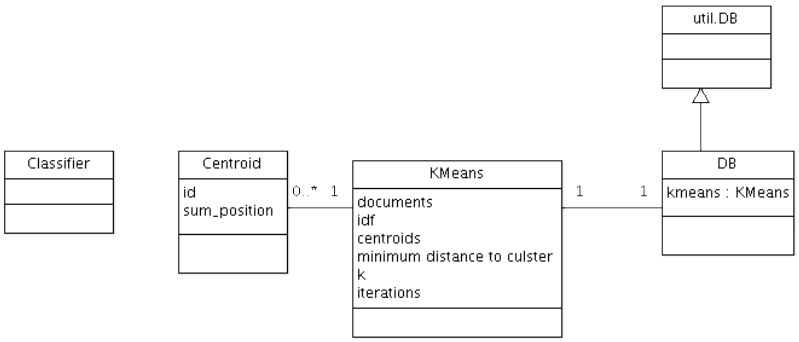
6.2 Organizer

The Organizer is a program that will organize the crawled documents according to the preferences of the user.

Classification and clustering algorithms will read the documents from the indexes that Nutch stores during the crawling. Then those documents will be classified according to the user's preferences. The resulting clustering or classification will be used during the "Focus Crawling" and the "Contextual Search".

6.2.1 K-Means clustering

The k-Means clustering part of our program, is designed as the following class diagram shows:



`KMeans` and `Centroid` are the most important and significant classes. `KMeans` class represents the k-means clustering algorithm, meanwhile `Centroid` represents each one of the centroids in the k-Means clustering algorithm. The most important attributes into the `KMeans` class are: `documents`, `idf`, `centroids`, `minimum distance to cluster`, `k` and `iterations` .

The `documents` attribute contains a list of all the document id's that are going to be clustered. The `idf` attribute is a Hashtable that contains the Inverse Document Frequency for each term. The `centroids` field contains a Hashtable with the ID's mapping their corresponding Centroid.

`minimum distance to cluster` attribute keeps the minimum known distance between each document and each centroid. The purpose of this attribute has been explained before.

`k` is the number of clusters of the clustering. `iterations` is the number of iterations of the k-Means algorithm.

Each centroid has its own ID, so that it can be identified. Furthermore, the Centroid contains a field called `sum_position`, that represents the sum of the positions of all the documents that it contains.

The other class that appears in the diagram is called DB. This class is used to interact with the database, so that the data can be retrieved and stored.

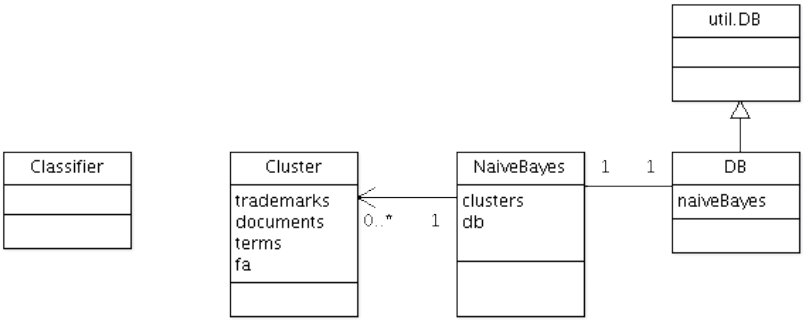
The `DB` class extends the `util.DB` class. This is because the DB class has access to the `KMeans` class, needed by some methods. Furthermore, this makes a separation between methods that are only going to be called while executing k-Means with the other methods.

When the clustering is stored in the database, not only the clusters with the assignation of their documents is stored. The `sum position`, `idf` and `minimum distance to cluster` fields will also be stored. This information is stored in order to prevent computing this information again.

A point to notice is that, when doing incremental clustering, `minimum distance to cluster` may change due to a change in IDF's, but the one stored in the database (computed using the old IDF) will be used anyway. We do so, because when centroids move, the `minimum distance to cluster` will increase, and it will be computed again using the correct IDF. Furthermore, IDF's don't change dramatically, so that the `minimum known distance` value will not be far from the `minimum known distance` computed with the new IDF. Moreover, it is generally accepted that computing approximations of idf values rather than their exact values is good enough in practice; see for example [24] for an explicit statement in this direction.

6.2.2 Naive Bayes classification

In this section we show the architecture of the part related to the Naïve Bayes classifier. The following class diagram contains the structure of our Naïve Bayes classifier:



The most important classes are `NaiveBayes` and `Cluster`. The attributes in `Cluster` are: `trademarks`, `documents`, `terms` and `fa`. The `trademarks` attribute is a HashSet of document id's, they represent the documents to be used as trademarks in that cluster; the trademarks are set manually by the user. The `documents` attribute contains the documents that have been assigned to this cluster. The `terms` attribute contains a Hashtable that maps each term to the number of trademarks where it appears. The `fa` attribute contains a Hashtable that maps each term to the probability for a trademark in the cluster to contain that term.

In the `NaiveBayes` class, the classification is performed, and the documents are assigned to their respective classes. At the end, the changes are saved in the database.

For Naive Bayes, we have implemented its `DB` class, which extends the `util.DB` class. The `DB` has access to the `NaiveBayes` instance, needed by some of the methods inside. Furthermore, this class helps to keep separated the methods used only in Naive Bayes from the rest of the database related methods.

Chapter 7

System Evaluation

7.1 Focused crawling

In this section we evaluate our focused crawling system. For this purpose we will compare two crawlings. In one case, we have used our focused crawling system. In the other one, we have used Nutch in the standard way, that is providing a manually created set of URLs to be used as seeds.

After performing this crawling, the crawling generated by our focused crawling system has a lot more variety of sites. Furthermore, the focused crawling system has indexed pages that are interesting to me, but I did not think of them to be used as seeds, thus they have not been added to the manual crawling index.

Note that, as explained in Chapter 4, the functionality to rate pages according to the interests of the user during the crawling is not working.

7.2 Organizer

7.2.1 Naïve Bayes Classification

In this section we evaluate the precision of our Naïve Bayes Classification algorithm. For this evaluation, we have executed a crawling in Nutch from a set of seeds of different topics, so that documents from different topics are crawled. These topics are: Computer Science (written as CS), Society (written as SO), Sports (written as S), Literature (written as L), Aviation (written as A) and Others (written as O).

The results of our classification algorithm are shown in the following table in the form of a confusion matrix. All the documents in each row R really belong to the topic R. And all the documents in each column C were classified as belonging to topic C. Note that all the documents that have been classified correctly, will appear in the diagonal.

	CS	SO	A	L	S	O
CS	89	0	0	0	0	0
SO	3	113	2	0	0	0
A	2	1	107	0	0	5
L	1	6	0	70	0	0
S	0	4	0	1	108	0
O	9	7	0	6	1	196

The results of the classification can be considered good, as 683 out of 731 documents (93.4 % of the documents) have been classified correctly.

7.2.2 k-Means clustering

In this section we evaluate our k-Means clustering algorithm.

Using an index based on our navigation index, we have performed clustering. The structure of the resulting clustering is shown in figure 7.1.

From the hierarchy in figure 7.1, we can see that most leaf clusters are useful, and most of them are of my interest. But the tree hierarchy is not shaped as we would expect to contain this set of topics. Anyways, the structure itself is not meaningful for the rest of the parts of the system, as they will only consider the leaf clusters.

As a consideration for the performance due to the usage of the "Minimum Known Distance" instead of the real distance in the K-Means algorithm, and using the `sum_position` vector for each centroid, we have performed another clustering to analyze the evolution during its iterations. In figure 7.2 we show the elapsed time and the number of changes for each iteration.

As you can observe, during the first iterations, the execution time is pretty hight, but as the algorithm advances, the execution time drops drastically. Up to the point that 14 out of 23 iterations are performed in less than 1/50 the time of the first iteration.

This phenomena can be explained by the fact that, the less changes occur, the least distance the centroids move. Thus, the "minimum known distance" values are kept near the value of the real distance. In addition, as the algorithm advances, less document assignation changes are necessary (remember that a cluster assignation change requires at least one "real distance" calculation). Those two factors, make the number of "real distance" calculations drop, thus saving CPU time. Furthermore, as less document reassignations occur, less computations about the new position of the centroids have to be executed: each time a document is reassigned, its position has to be subtracted from the `sum_position` vector in the old centroid, and its position has to be added to the `sum_position` vector of the new centroid. Moreover, if a centroid does not move during the whole iteration, its new position will not be recomputed after the iteration.

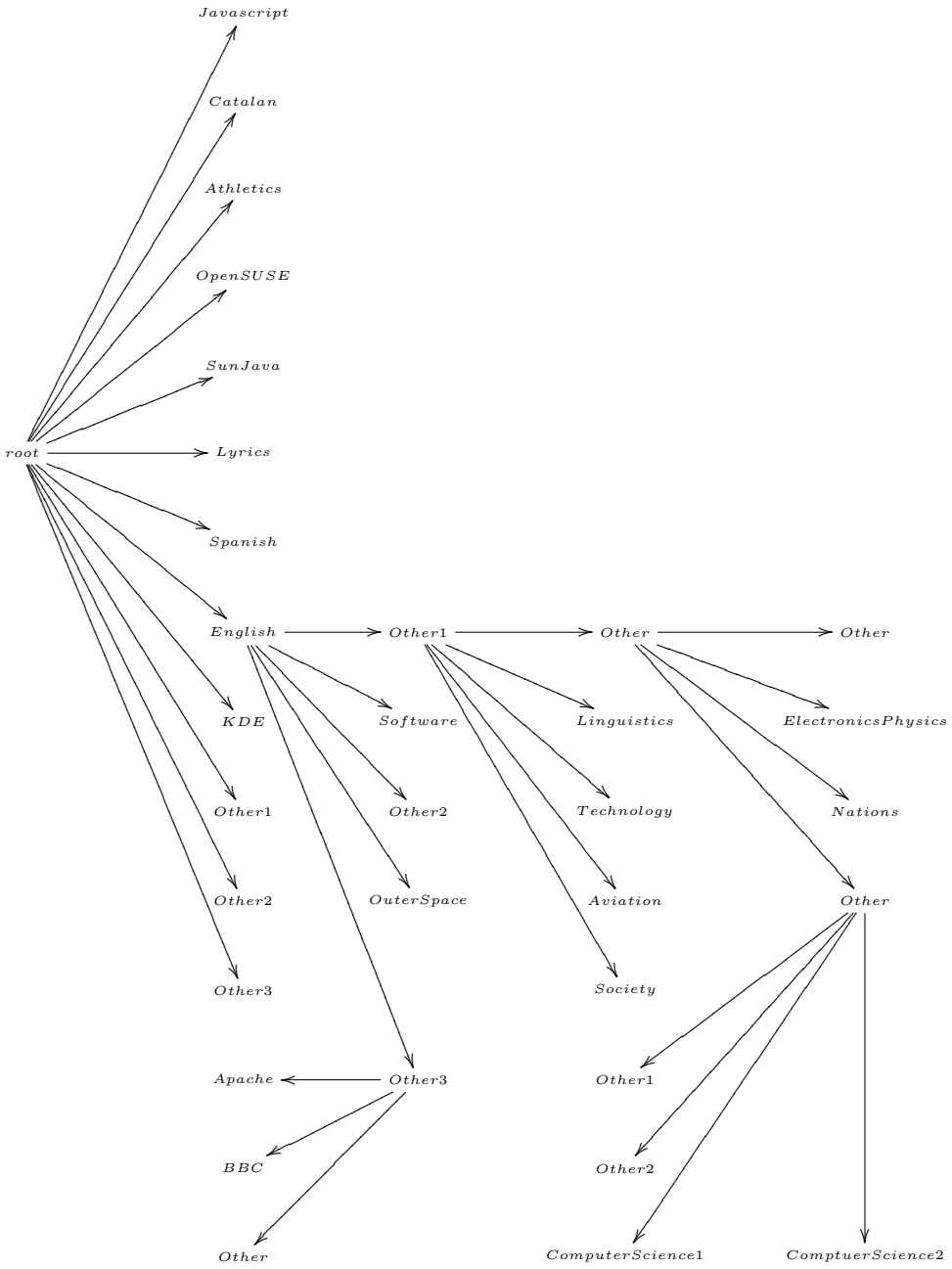


Figure 7.1: Result of the clustering algorithm

Iteration	Number of changes	Elapsed Time (seconds)	% of time
1	2020	1843.718	100
2	622	575.532	31,22
3	249	240.233	13,03
4	110	115.842	6,28
5	48	64.354	3,49
6	47	60.599	3,29
7	46	60.616	3,29
8	34	49.621	2,69
9	8	25.185	1,37
10	9	22.627	1,23
11	9	21.232	1,15
12	7	21.244	1,15
13	13	26.441	1,43
14	11	23.698	1,29
15	4	20.092	1,09
16	2	12.861	0,7
17	2	14.883	0,81
18	2	9.014	0,49
19	3	13.504	0,73
20	3	23.616	1,28
21	1	11.443	0,62
22	1	9.309	0,5
23	0	8.58	0,47

Figure 7.2: Analyzis of the k-Means clustering execution time

7.3 Nutch Search

In this section we evaluate the Nutch Search part of our program. For this purpose we will perform some searches in different contexts and analyze the result of our queries.

First we will search for "pitch". The word "pitch" has different meanings depending on the subject, but we are interested in the meaning of pitch related to aviation. In the next table we show the results for this search for two different cases: when "aviation" is the navigation context, and when the navigation context is "sports". For this case, we have used the Naïve Bayes algorithm as classification algorithm.

Sports	**Aviation**
BBC - Rock/Indie Review - Panda Bear, Person Pitch	Propeller - Wikipedia
Blade Pitch - Wikipedia	Helicopter flight controls - Wikipedia
BBC SPORT — Football	BBC - Rock/Indie Review - Panda Bear
Helicopter flight controls - Wikipedia	Autogyro - Wikipedia
BBC SPORT — Football — FA Cup ...	Blade Pitch - Wikipedia
BBC SPORT — Football — Non League ...	Angle of attack - Wikipedia
University of Delaware Athletics and ...	Fixed-wing aircraft - Wikipedia
BBC SPORT — Football	Wing - Wikipedia
Propeller - Wikipedia	Flight - Wikipedia
BBC SPORT — Football	Helicopter - Wikipedia

As shown, in the top ten results for the "Aviation" context, 9 out of 10 web pages of the result are related to "Aviation", (only the page "BBC - Rock/Indie Review - Panda Bear" is not related to aviation). By contrast, when the context is "Sports", only 3 pages are related to aviation and 6 are related to "Sports". Note that our search context modifications only modify the ranking of the result, not the set of pages of the result. Therefore, if the user is not interested in the topic of his/her context, he can search over the pages with lower ranking.

Moreover, when the search "pitch" is performed in Google, it will not return any page related to aviation in the 20 first results. This happens because Google searches for this word over the whole internet, not in the pages of interest for the user.

Furthermore, if the user has been visiting pages related to aviation, but now wants to visit pages related to sports, he/she can always make his/her search more precise, for example search for "pitch football". For the search "pitch football" in the "Aviation" context the result would be:

1. BBC SPORT — Football
2. BBC SPORT — Football — FA Cup — Histon 1-2 Swansea
3. BBC SPORT — Football — Non League — FA Trophy second round results
4. BBC - Rock/Indie Review - Panda Bear, Person Pitch
5. University of Delaware Athletics and Sports Information
6. Powered hang glider - Wikipedia
7. Paragliding - Wikipedia

As you can see, the system will return pages related to sports, as there are very few pages related to aviation where the word "football" appears.

In the next case, we will perform the same query ("pitch"), but using the k-Means algorithm instead of Naïve Bayes. Moreover, we will compare the results obtained while surfing over "BBC News" pages, and the results obtained while surfing among "aviation" pages.

BBC News	**Aviation**
BBC - Rock/Indie Review - Panda Bear, Person Pitch	Helicopter flight controls - Wikipedia
Blade Pitch - Wikipedia	Autogyro - Wikipedia
BBC SPORT — Football	BBC - Rock/Indie Review - Panda Bear
BBC SPORT — Football — My Club... - Wikipedia	Blade Pitch - Wikipedia
BBC SPORT — Football	Propeller - Wikipedia
Helicopter flight controls - Wikipedia	BBC SPORT — Football
BBC SPORT — Football ...	Swashplate (helicopter) - Wikipedia
Propeller - Wikipedia	Juan de la Cierva - Wikipedia
BBC SPORT — Football — ...	Helicopter flight controls - Wikipedia
BBC SPORT — Football	Helicopter rotor - Wikipedia

As can be observed, when the k-Means algorithm is used to analyze the context, the result is also good.

Also remark that when query "pitch" is submitted to Google, it will return no page related to aviation in the top twenty results.

Chapter 8

Conclusions and Future Work

Standard web searchers consider very little information about the user. We thought that analyzing the clickstream would give meaningful information to the system, thus making a more personalized search engine.

In this work, we have built a system that is capable to find out the interests of a single user, so that searches are focused to his/her topics of interests and his/her navigation context. Usually, each user has his/her own interests, but some groups of people do share their interests. In many work groups (such as a small company, or a department within a large company, or a team working on a single project, or students in the same course) people are sharing an important part of their interests. Thus, making our system capable to deal with the interests of groups of users instead of a single user would make it useful for some groups of people too.

Regarding our clustering system, note that most users are not monolingual, but multilingual. If the user visits pages in different languages, our system will classify them in different clusters, although they refer to the same topic. To prevent this behavior, a translation to one language (i.e. to English language) could be applied to all the pages before the clustering algorithm was applied. Even if the automatic translation is far from perfect from a human point of view, it would perhaps give a sufficient idea of the topic of the document so that the quality of our clustering would improve.

As for crawling, our system performs a focused crawling, thus retrieving only the documents that can be of interest for the user. But our crawling system has a limited capacity, as it is to be executed on a single machine which is also used for a lot more purposes. This makes the crawling too limited. A possible solution for this would be to execute the queries, that have been modified according to the navigation context, in a large-scale web searcher which does crawl the whole web (e.g. Google) instead of our search engine. This search would consider a much wider range of pages, but would still be focused considering the interests of the user and his/her navigation context.

Bibliography

[1] Serge Abiteboul, Mihai Preda, and Gregory Cobena. Adaptive on-line page importance computation. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 280–290, New York, NY, USA, 2003. ACM.

[2] Ricardo Baeza Yates and Berthier Ribeiro Nieto. *Modern Information Retrieval*. Addison Wesley, 1998.

[3] Matthias Bender, Sebastian Michel, Christian Zimmer, and Gerhard Wikum. Bookmark-driven query routing in peer-to-peer web search. *SIGIR Workshop on Peer-to-Peer Information Retrieval. 2004*.

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems*, pages 107–117, 1998.

[5] G.C.Garriga. Summarizing sequential data with closed partial orders. In *2005 SIAM International Conference on Data Mining (SDM'05)*.

[6] Panagiotis Giannikopoulos, Iraklis Varlamis, and Magdalini Eirinaki. Mining frequent generalized patterns for web personalization. *MSODA 2008*, 2008.

[7] Zoltan Gyongyi, Hector Garcia-Molina, and Jan Pedersen. Combating web spam with trustrank. Technical Report 2004-17, Stanford InfoLab, March 2004.

[8] Martin Halvey, Mark T. Keane, and Barry Smyth. Time-based segmentation of log data for user navigation prediction in personalization. *International Conference on Web Intelligence (WI'05)*.

[9] Shun Hattori, Taro Tezuka, and Katsumi Tanaka. Context-aware query refinement for mobile web search. *Symposium on Applications and the Internet Workshops (SAINTW'07)*.

[10] Taher H. Haveliwala. Topic-sensitive pagerank. In *Eleventh International World Wide Web Conference (WWW 2002)*, 2002.

[11] http://lucene.apache.org/nutch/.

[12] http://www.searchenginejournal.com/wp-content/uploads/2008/12/competesearchmarketshare.jpg.

[13] Witten I.H. and Frank E. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufman Publishers, 2000.

[14] Hernández Orallo J., Ramírez Quintana M.J., and Ferri Ramírez C. *Introducción a la Minería de Datos*. Prentice Hall, 2004.

[15] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *9th ACM-SIAM Symposium on Discrete Algorithms, 1998*.

[16] Steve Lawrence. Context in web search. *IEEE Data Engineering Bulletin*, 23:25–32, 2000.

[17] Fang Liu, Clement Yu, and Weiyi Meng. Personalized web search by mapping user queries to categories. *CIKM'02*.

[18] Fang Liu, Clement Yu, and Weiyi Meng. Personalized web search for improving retrieval effectiveness.

[19] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *An Introduction to Information Retrieval*. Cambridge University Press, to appear, 2009.

[20] Massimo Marchiori. The quest for correct information on the web: Hyper search engines. In *Proceedings of the Sixth International World Wide Web Conference (WWW6), 1997*.

[21] Zdravko Markov and Daniel T. Larose. *Data mining the web*. Wiley Interscience, 2007.

[22] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media Inc., 2007.

[23] Hastie T., Tibshirani R., and Friedman J. *The elements of statistical learning. Data mining, inference and prediction*. Springer, 2001.

[24] Hans Friedrich Witschel. Global term weights in distributed environments. *Inf. Process. Manage.*, 44(3):1049–1061, 2008.

[25] Ian. H Witten, Alistair. Moffat, and Timothy C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 1999.

Appendix: System Manual

Organizer

To start using the PS system, you first need to collect some navigation history. NTExt, the Firefox Extension, will be used to log the navigation history.

Once some navigation history has been collected (over 1000 pages visited), we will run CrawlUnknownHistoryURLs, that will collect all the URLs in the navigation history log. Then, a crawling using those URLs will be performed.

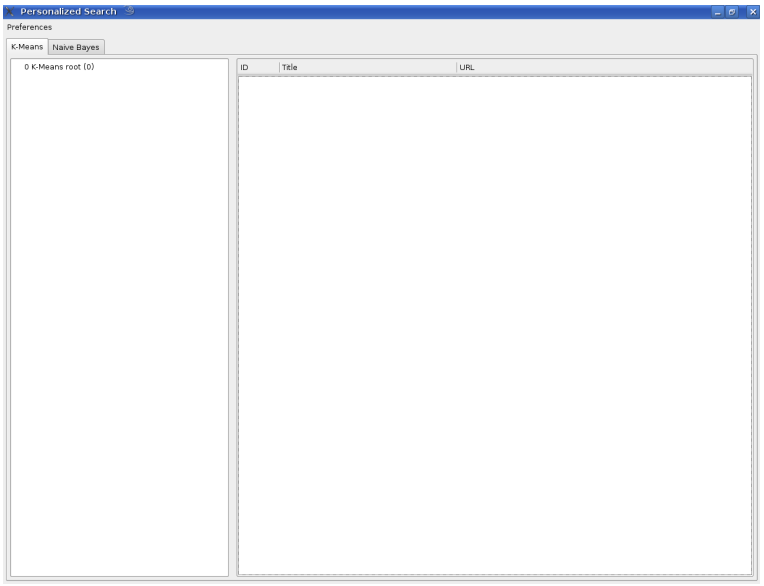
The next step is to run the program ClusterHistoryURLs. This program will do a clustering with all the URLs that the user visited and have been crawled. From this clustering, the nearest to the centroid URLs will be retrieved. Those URLs will be used as seeds for the crawling.

Once another crawling has been performed, the user can start the Organizer, and so organize the documents provided by the Nutch Crawler.

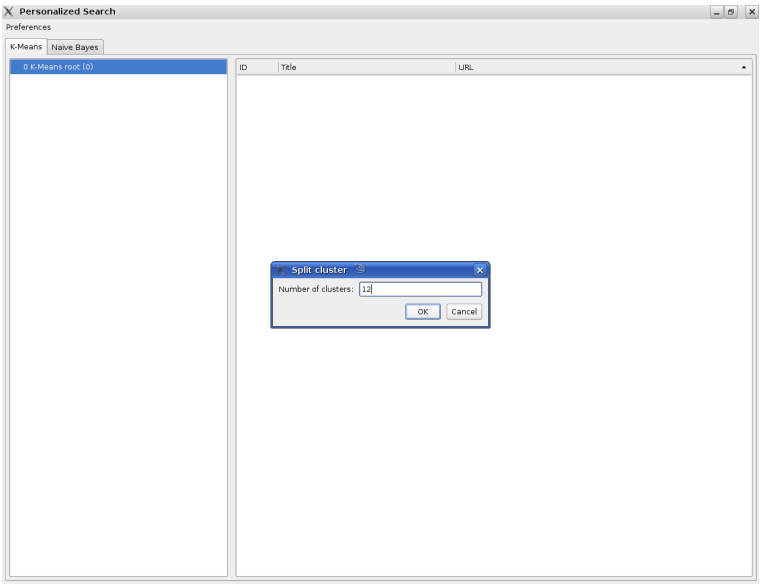
To switch between the k-Means algorithm and the Naïve Bayes algorithm to be used as the organizing algorithm during the crawling and the searches, go to "Preferences → Organizing Algorithm". Then choose the algorithm you want.

Using the k-Means clustering algorithm

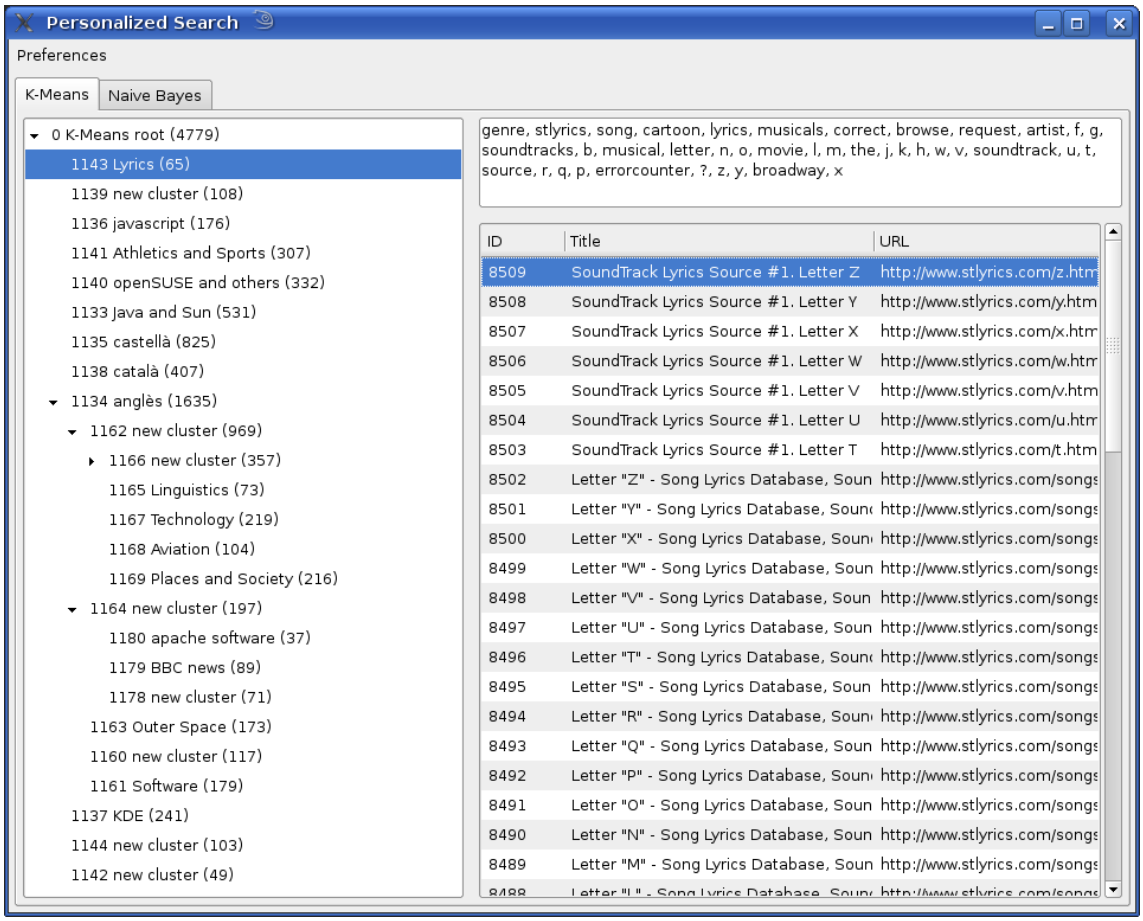
In this section we show how you can start using the k-Means clustering algorithm. Once you have started the Organizer, you will get the following screen:



To organize the documents, we will split the root cluster into 12 sub-clusters.



Then, the clusters that contain documents related to many different topics will be splitted into more clusters. After some clusterings, the hierarchy will be as follows:

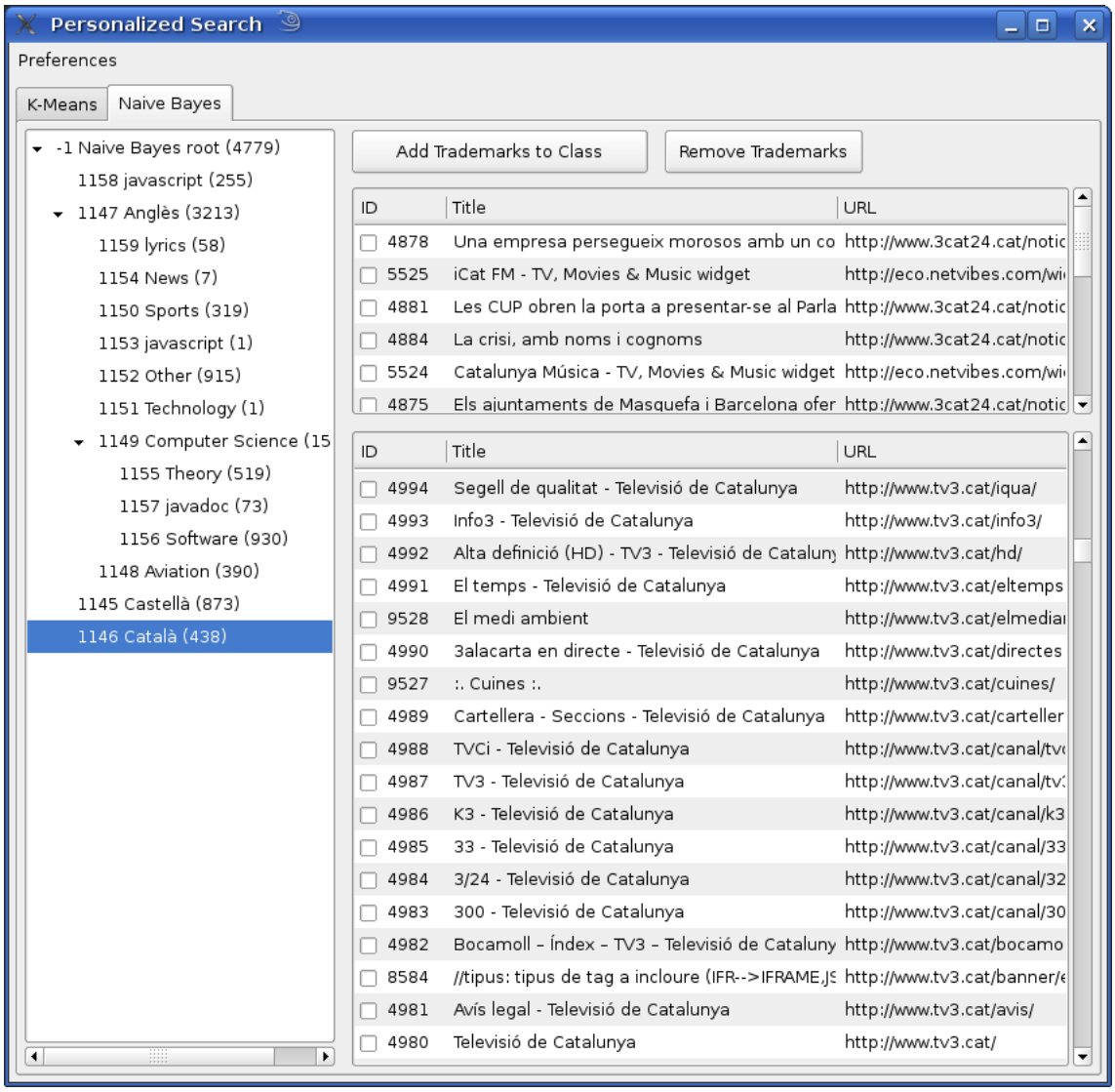


On the left of the window there is a tree that represents the hierarchy of the clustering. Each cluster is represented as: ID name (number of documents in the cluster), for example, cluster "1143 Lyrics (65)" means that this cluster has the ID 1143, is called "Lyrics" and contains 65 documents. Furthermore, on the right of the screen there are two boxes. In the box on the top, the most meaningful terms for the currently selected cluster appear. And in the box on the bottom, the list of documents for the currently selected cluster appear.

Using Naïve Bayes Classification algorithm

In this section we will introduce you to the usage of the Naïve Bayes classifier included in the PS system.

The main window in the Naïve Bayes classifier looks like the following screenshot:

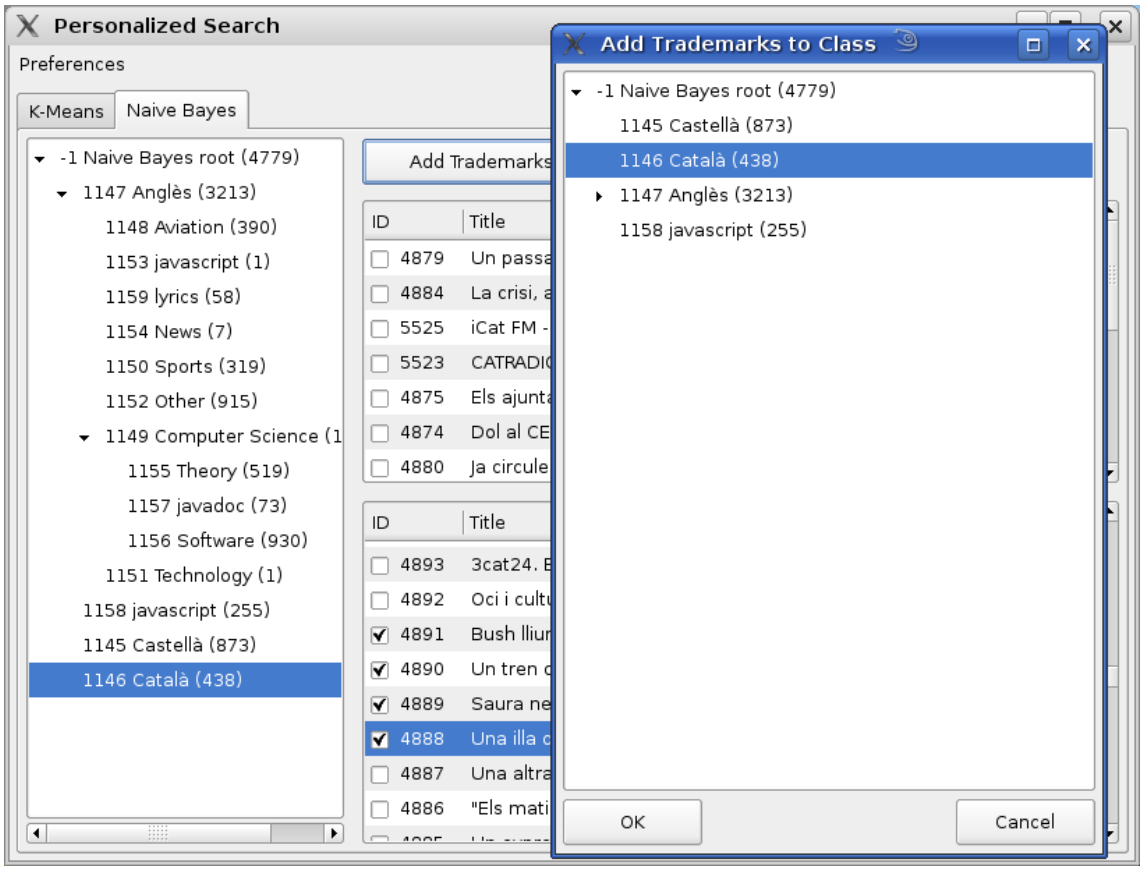


On the left of the window you can see the hierarchy of the classes (like in k-Means). On the right of the window there are two boxes. The box on the bottom contains a list of all the documents in the currently selected class. The box on the top contains a list of all the documents that are trademarks for the currently selected class.

To add a children to a class, you just need to right-click on that class and select the "Add child" option. Then you will be prompted for a name for that class. Write the name you desire and push the "Ok" button. Then, you will need to specify a set of trademarks for this class.

To add trademarks to a class, select the documents you want to set as trademarks and push the "Add trademarks to Cluster", then a dialog will appear. In this dialog you have to select the class where you desire to add the trademarks.

Finally, push the “Ok” button. This procedure is performed in the following screenshot:



Once you have set the trademarks to the classes, you can start the classification algorithm. To start the classification algorithm, right-click on the class from which you want to start the classification algorithm; then select the “Re-classify” option. Then all the document in this class will be re-classified among its predecessor classes recursively.

Search

To start performing searches, the “UpdateContext” program has to be started. This program will insert the changes of the navigation context into the database.

Then, when the user performs a query against our system, this search will be modified to add information about the navigation context.

Navigation Tips

Navigation Tips searches for patterns in the click-stream of the user. From these patterns, some rules are retrieved. These rules will be used by NTExt (the Firefox Extension), to provide some navigation tips according to the navigation context and the rules retrieved from the click-stream.

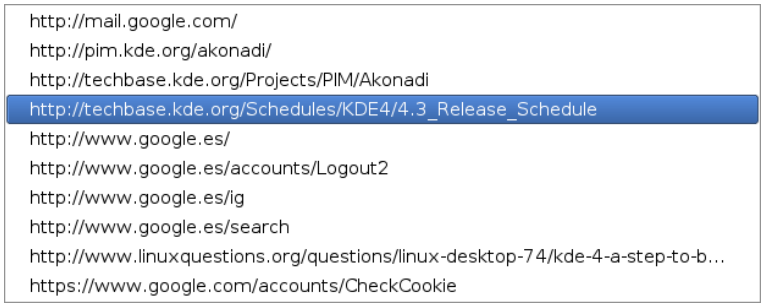
Once the NTExt has been installed, it will start logging the pages visited in a file called history.txt. When this file is long enough, typically around 1,000 lines, you can execute the historyAnalysis program. This program will retrieve some rules from the history.txt, and will save them in the rules.txt file.

When the rules.txt file contains rules, these will be used by NTExt to provide navigation tips to the user.

In the next case, we already have a long history.txt file, and the rules.txt file has been updated by the historyAnalysis program according to the current history.txt file.

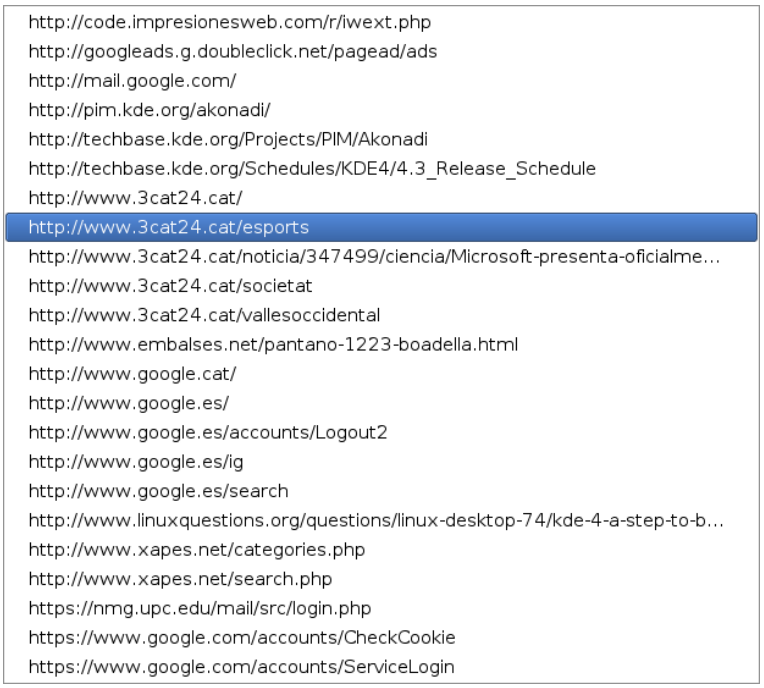
When clicking in Firefox, a list of navigation tips will be provided.

Consider the next example: we have just started firefox, so only the home page (in this case http://www.google.es/) has been loaded. When I click the icon, I get the next list of navigation tips:



The first option is http://mail.google.com/, which is the mail service that I use most. And the next ones are pages that I've been visiting during the latest days.

If I visit http://www.3cat24.cat/, the navigation tips list will change, as I have changed my context:



As you can see, many new pages have appeared among the navigation tips. The ones that I use more often are: http://www.3cat24.cat/esports, http://www.3cat24.cat/vallesoccidental, http://www.3cat24.cat/societat. Furthermore, http://www.3cat24.cat/Microsoft-presenta-oficialment... has also appeared in the navigation tips because I have visited this page several times during the latest days. If I spend some days without visiting this page, the historyAnalysis program will detect this change in my behavior, and will delete this page from the rules.